John von Neumann Institute for Computing

Memory Debugging of MPI-Parallel Applications in Open MPI

Rainer Keller, Shiqing Fan, Michael Resch

published in

Parallel Computing: Architectures, Algorithms and Applications ,
C. Bischof, M. Bücker, P. Gibbon, G.R. Joubert, T. Lippert, B. Mohr, F. Peters (Eds.),
John von Neumann Institute for Computing, Jülich,
NIC Series, Vol. **38**, ISBN 978-3-9810843-4-4, pp. 517-523, 2007.
Reprinted in: Advances in Parallel Computing, Volume **15**,
ISSN 0927-5452, ISBN 978-1-58603-796-3 (IOS Press), 2008.

http://www.fz-juelich.de/nic-series/volume38

Memory Debugging of MPI-Parallel Applications in Open MPI

Rainer Keller, Shiqing Fan, and Michael Resch

High-Performance Computing Center, University of Stuttgart,
E-mail: {keller, fan, resch}@hlrs.de

In this paper we describe the implementation of memory checking functionality based on instrumentation using `valgrind`. The combination of valgrind based checking functions within the MPI-implementation offers superior debugging functionality, for errors that otherwise are not possible to detect with comparable MPI-debugging tools. The functionality is integrated into Open MPI as the so-called `memchecker`-framework. This allows other memory debuggers that offer a similar API to be integrated. The tight control of the user's memory passed to Open MPI, allows not only to find application errors, but also helps track bugs within Open MPI itself.

We describe the actual checks, classes of errors being found, how memory buffers internally are being handled, show errors actually found in user's code and the performance implications of this instrumentation.

1 Introduction

Parallel programming with the distributed memory paradigm using the Message Passing Interface MPI[1] is often considered as an error-prone process. Great effort has been put into parallelizing libraries and applications using MPI. However when it comes to maintaining the software, optimizing for new hardware or even porting the code to other platforms and other MPI implementations, the developers face additional difficulties[2]. They may experience errors due to implementation-defined behaviour, hard-to-track timing-critical bugs or deadlocks due to communication characteristics of the MPI-implementation or even hardware dependent behaviour. One class of bugs, that are hard-to-track are memory errors, specifically in non-blocking communication.

This paper introduces a debugging feature based on instrumentation functionality offered by `valgrind`[3], that is being employed within the Open MPI-library. The user's parameters, as well as other non-conforming MPI-usage and hard-to-track errors, such as accessing buffers of active non-blocking operations are being checked and reported. This kind of functionality would otherwise not be possible within traditional MPI-debuggers based on the PMPI-interface.

This paper is structured as follows: Section 2 gives an introduction into the design and implementation, Section 3 shows the performance implications, Section 4 shows the errors, that are being detected. Finally, Section 5 gives a comparison of other available tools and concludes the paper with an outlook.

2 Design and Implementation

The tool suite `valgrind`[3] may be employed on static and dynamic binary executables on `x86`/`x86_64`/`amd64`- and PowerPC32/64-compatible architectures. It operates by intercepting the execution of the application on the binary level and interprets and instruments

the instructions. Using this instrumentation, the tools within the `valgrind`-suite then may deduce information, e. g. on the allocation of memory being accessed or the definedness of the content being read from memory. Thereby, the `memcheck`-tool may detect errors such as buffer-overruns, faulty stack-access or allocation-errors such as dangling pointers or double frees by tracking calls to `malloc`, `new` or `free`. Briefly, the tool `valgrind` shadows each byte in memory: information is kept whether the byte has been allocated (so-called A-Bits) and for each bit of each byte, whether it contains a defined value (so-called V-Bits).

As has been described on the web-page[4] for MPIch since version 1.1, `valgrind` can be used to check MPI-parallel applications. For MPIch-1[5] `valgrind` has to be declared as debugger, while for Open MPI, one only prepends the application with `valgrind` and any valgrind-parameters, e. g. `mpirun -np 8 valgrind --num_callers=20 ./my_app inputfile`.

As described, this may detect memory access bugs, such as buffer overruns and more, but also by knowledge of the semantics of calls like `strncpy`. However, `valgrind` does not have any knowledge of the semantics of MPI-calls. Also, due to the way, how `valgrind` is working, errors due to undefined data may be reported late, way down in the call stack. The original source of error in the application therefore may not be obvious.

In order to find MPI-related hard-to-track bugs in the application (and within Open MPI for that matter), we have taken advantage of an instrumentation-API offered by `memcheck`. To allow other kinds of memory-debuggers, such as `bcheck` or Totalview's memory debugging features[6], we have implemented the functionality as a module into Open MPI's Modular Component Architecture[7]. The module is therefore called `memchecker` and may be enabled with the configure-option `--enable-memchecker`.

The instrumentation for the `valgrind`-parser uses processor instructions that do not otherwise change the semantics of the application. By this special instruction preamble, `valgrind` detects commands to steer the instrumentation. On the x86-architecture, the right-rotation instruction `ror` is used to rotate the 32-bit register `edi`, by 3, 13, 29 and 19, aka 64-Bits, leaving the same value in `edi`; the actual command to be executed is then encoded with an register-exchange instruction (`xchgl`) that replaces a register with itself (in this case `ebx`):

```
#define __SPECIAL_INSTRUCTION_PREAMBLE                          \
            "roll $3,  %%edi ; roll $13, %%edi\n\t"     \
            "roll $29, %%edi ; roll $19, %%edi\n\t"     \
            "xchgl %%ebx, %%ebx\n\t"
```

In Open MPI objects such as communicators, types and requests are declared as pointers to structures. These objects when passed to MPI-calls are being immediately checked for definedness and together with `MPI_Status` are checked upon exit[a]. Memory being passed to Send-operations is being checked for accessibility and definedness, while pointers in Recv-operations are checked for accessibility, only.

Reading or writing to buffers of active, non-blocking Recv-operations and writing to buffers of active, non-blocking Send-operations are obvious bugs. Buffers being passed to non-blocking operations (after the above checking) is being set to undefined within the MPI-layer of Open MPI until the corresponding completion operation is issued. This set-

[a] E. g. this showed up uninitialized data in derived objects, e. g. communicators created using `MPI_Comm_dup`

ting of the visibility is being set independent of non-blocking `MPI_Isend` or `MPI_Irecv` function. When the application touches the corresponding part in memory before the completion with `MPI_Wait`, `MPI_Test` or multiple completion calls, an error message will be issued. In order to allow the lower-level MPI-functionality to send the user-buffer as fragment, the so-called lower layer Byte Transfer Layer (BTLs) are adapted to set the fragment in question to accessible and defined, as may be seen in Fig. 1. Care has been taken to handle derived datatypes and it's implications.

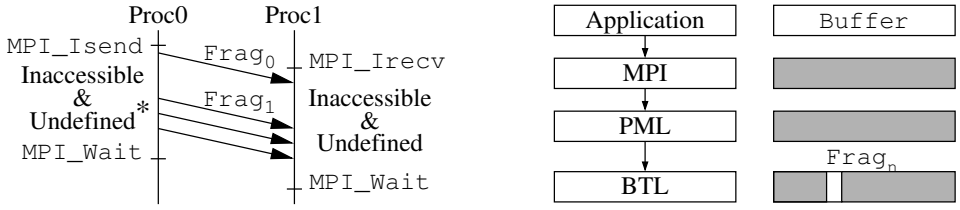

Figure 1. Fragment handling to set accessibility and definedness

For Send-operations, the MPI-1 standard also defines, that the application may not access the send-buffer at all (see[1], p. 30). Many applications do not obey this strict policy, domain-decomposition based applications that communicate ghost-cells, still read from the send-buffer. To the authors' knowledge, no existing implementation requires this policy, therefore the setting to undefined on the Send-side is only done with strict-checking enabled (see Undefined* in Fig. 1).

3 Performance Implications

Adding instrumentation to the code does induce a slight performance hit due to the assembler instructions as explained above, even when the application is not run under `valgrind`. Tests have been done using the Intel MPI Benchmark (IMB), formerly known as Pallas MPI Benchmark (PMB) and the BT-Benchmark of the NAS parallel benchmark suite (NPB) all on the `dgrid`-cluster at HLRS. This machine consists of dual-processor Intel Woodcrest, using Infiniband-DDR network with the OpenFabrics stack.

For IMB, two nodes were used to test the following cases: with&without `--enable-memchecker` compilation and with `--enable-memchecker` but without MPI-object checking (see Fig. 2) and with&without `valgrind` was run (see Fig. 3). We include the performance results on two nodes using the PingPong test. In Fig. 2 the measured latencies (left) and bandwidth (right) using Infiniband (not running with `valgrind`) shows the costs incurred by the additional instrumentation, ranging from 18 to 25% when the MPI-object checking is enabled as well, and 3-6% when memchecker is enabled, but no MPI-object checking is performed. As one may note, while latency is sensitive to the instrumentation added, for larger packet-sizes, it is hardly noticeable anymore (less than 1% overhead). Figure 3 shows the cost when additionally running with `valgrind`, again without further instrumentation compared with our additional instrumentation applied, here using TCP connections employing the IPoverIB-interface.

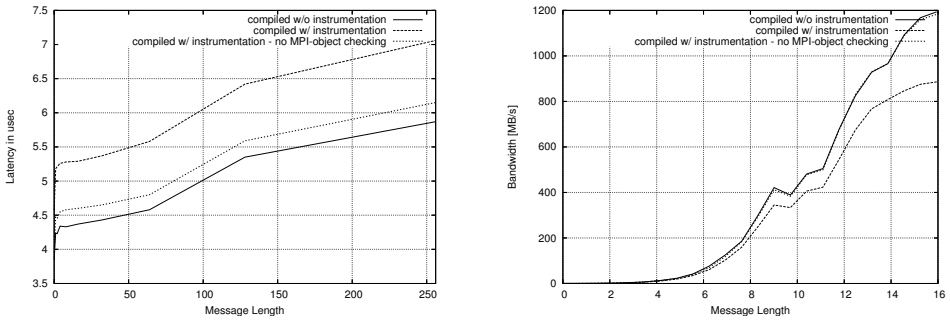

Figure 2. Latencies and bandwidth with&without memchecker-instrumentation over Infiniband, running without `valgrind`.

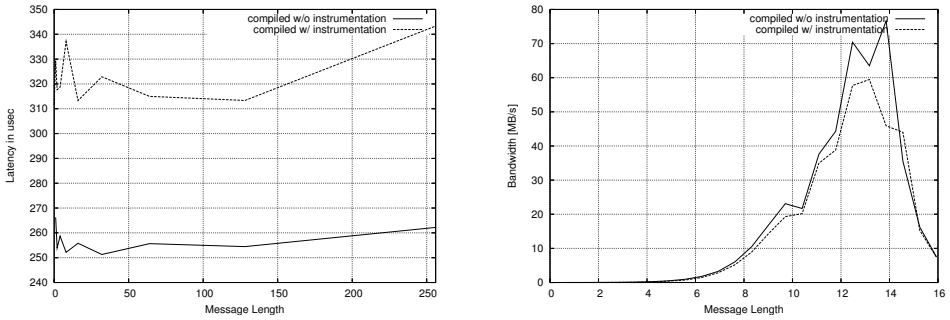

Figure 3. Latencies and bandwidth with&without memchecker-instrumentation using IPoverIB, running with `valgrind`.

The large slowdown of the MPI-object checking is due to the tests of every argument and its components, i. e. the internal data structures of an `MPI_Comm` consist of checking the definedness of 58 components, checking an `MPI_Request` involves 24 components, while checking `MPI_Datatype` depends on the number of the base types.

The BT-Benchmark has several classes, which have different complexity, and data size. The algorithm of BT-Benchmark solves three sets of uncoupled systems of equations, first in the x, then in the y, and finally in the z direction. The tests are done with sizes Class A and Class B. Figure 4 shows the time in seconds for the BT Benchmark. The Class A (size of 64x64x64) and Class B (size of 102x102x102) test was run with the standard parameters (200 iterations, time-step dt of 0.0008).

Again, we tested Open MPI in the following three cases: Open MPI without memchecker component, running under `valgrind` with the memchecker component disabled and finally with `--enable-memchecker`.

As may be seen and is expected this benchmark does not show any performance implications whether the instrumentation is added or not. Of course due to the large memory requirements, the execution shows the expected slow-down when running under valgrind, as every memory access is being checked.

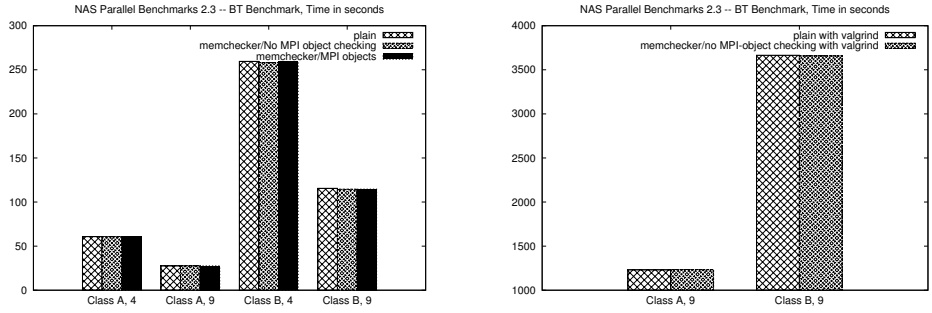

Figure 4. Time of the NPB/BT benchmark for different classes running without (left) and with (right) `valgrind`.

4 Detectable Error Classes and Findings in Actual Applications

The kind of errors, detectable with a memory debugging tool such as `valgrind` in conjunction with instrumentation of the MPI-implementation are:

- Wrong input parameters, e. g. undefined memory passed into Open MPI[b]:

```
char * send_buffer;
send_buffer = (char *) malloc (5);
memset (send_buffer, 0, 5);
MPI_Send (send_buffer, 10, MPI_CHAR, 1, 0, MPI_COMM_WORLD);
```

- Wrong input parameters, wrongly sized receive buffers:

```
char * recv_buffer;
recv_buffer = (char*) malloc(SIZE-1);
memset (buffer, SIZE-1, 0);
MPI_Recv (buffer, SIZE, MPI_CHAR, ..., &status);
```

- Uninitialized input buffers:

```
char * buffer;
buffer = (char *) malloc (10);
MPI_Send (buffer, 10, MPI_INT, 1, 0, MPI_COMM_WORLD);
```

- Usage of the uninitialized `MPI_ERROR`-field of `MPI_Status`[c]:

```
MPI_Wait (&request, &status);
if(status.MPI_ERROR != MPI_SUCCESS)
    return ERROR;
```

- Writing into the buffer of active non-blocking Send or Recv-operation or persistent communication:

[b] This could be found with a BTL such as TCP, however not with any NIC using RDMA.
[c] The MPI-1 standard defines the `MPI_ERROR`-field to be undefined for single-completion calls such as `MPI_Wait` or `MPI_Test` (p. 22).

```
int buf = 0;
MPI_Request req;
MPI_Status status;
MPI_Irecv (&buf, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, &req);
buf = 4711;                    /* Will produce a warning */
MPI_Wait (&req, &status);
```

- Read from the buffer of active non-blocking Send-operation in strict-mode:

```
int inner_value = 0, shadow = 0;
MPI_Request req;
MPI_Status status;
MPI_Isend (&shadow, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, &req);
inner_value += shadow;         /* Will produce a warning */
MPI_Wait (&req, &status);
```

- Uninitialized values, e. g. MPI-objects from within Open MPI.

During the course of development, several software packages have been tested with the memchecker functionality. Among them problems showed up in Open MPI itself (failed in initialization of fields of the status copied to user-space), an MPI testsuite[8], where tests for the `MPI_ERROR` triggered an error. In order to reduce the number of false positives Infiniband-networks, the `ibverbs`-library of the OFED-stack[9] was extended with instrumentation for buffer passed back from kernel-space.

5 Conclusion

We have presented an implementation of memory debugging features into Open MPI, using the instrumentation of the `valgrind`-suite. This allows detection of hard-to-find bugs in MPI-parallel applications, libraries and Open MPI itself[2]. This is new work, up to now, no other debugger is able to find these kind of errors.

With regard to related work, debuggers such as Umpire[10], Marmot[11] or the Intel Trace Analyzer and Collector[2], actually any other debugger based on the Profiling Interface of MPI, may detect bugs regarding non-standard access to buffers used in active, non-blocking communication without hiding false positives of the MPI-library itself.

In the future, we would like to extend the checking for other MPI-objects, extend for MPI-2 features, such as one-sided communication, non-blocking Parallel-IO access and possibly other error-classes.

References

1. Message Passing Interface Forum, *MPI: A Message Passing Interface Standard*, (1995). `http://www.mpi-forum.org`
2. J. DeSouza, B. Kuhn and B. R. de Supinski, *Automated, scalable debugging of MPI programs with Intel message checker*, in: Proc. 2nd International Workshop on Software Engineering for High Performance Computing System Applications, vol. **4**, pp. 78–82, (ACM Press, NY, 2005).

3. J. Seward and N. Nethercote, *Using Valgrind to detect undefined value errors with bit-precision*, in: Proc. USENIX'05 Annual Technical Conference, Anaheim, CA, (2005).
4. R. Keller, *Using Valgrind with MPIch*, Internet. `http://www.hlrs.de/people/keller/mpich_valgrind.html`
5. W. Gropp, E. Lusk, N. Doss and A. Skjellum, *A high-performance, portable implementation of the MPI message passing interface standard*, Parallel Computing, **22**, 789–828, (1996).
6. Totalview Memory Debugging capabilities, Internet. `http://www.etnus.com/TotalView/Memory.html`
7. T. S. Woodall, R. L. Graham, R. H. Castain, D. J. Daniel, M. W. Sukalski, G. E. Fagg, E. Gabriel, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett and A. Lumsdaine, *Open MPI's TEG Point-to-Point communications methodology: comparison to existing implementations*, in: Proc. 11th European PVM/MPI Users' Group Meeting, D. Kranzlmüller, P. Kacsuk, and J. J. Dongarra, (Eds.), vol. **3241** of *Lecture Notes in Computer Science (LNCS)*, pp. 105–111, (Springer, 2004).
8. R. Keller and M. Resch, *Testing the correctness of MPI implementations*, in: Proc. 5th Int. Symp. on Parallel and Distributed Computing (ISDP), Timisoara, Romania, (2006).
9. *The OpenFabrics project webpage*, Internet, (2007). `https://www.openfabrics.org`
10. J. S. Vetter and B. R. de Supinski, *Dynamic software testing of MPI applications with Umpire*, in: Proc. SC'00, (2000).
11. B. Krammer, M. S. Müller and M. M. Resch, *Runtime checking of MPI applications with Marmot*, in: Proc. PARCO'05, Malaga, Spain, (2005).